Stabilisation Instantanée Probabiliste

Karine Altisen, Stéphane Devismes

Stabilisation Instantanée Probabiliste

Karine Altisen[1] et Stéphane Devismes[1]

[1] *VERIMAG, Université Grenoble-Alpes*

Cet article est un résumé étendu de [1], où nous introduisons la *stabilisation instantanée probabiliste*. Cette propriété nous permet, en particulier, de concevoir des algorithmes distribués pour réseaux anonymes ayant de fortes propriétés de tolérance aux pannes transitoires. Un algorithme instantanément stabilisant probabiliste satisfait la sûreté de sa spécification *immédiatement* après que les pannes transitoires aient cessé ; cependant il n'assure la vivacité de sa spécification que *presque sûrement*. Nous illustrons cette nouvelle propriété en proposant deux algorithmes instantanément stabilisants probabilistes d'élection avec garantie de service pour réseaux anonymes, ce problème n'ayant pas de solution déterministe.

Mots-clés : stabilisation instantanée, algorithmes probabilistes, élection

1 Introduction

L'*autostabilisation* est un paradigme général permettant de concevoir des algorithmes distribués tolérant les fautes transitoires. Une faute transitoire est une panne non-définitive qui altère le contenu du composant du réseau (processus ou canal de communication) où elle se produit. En supposant que les fautes transitoires n'altèrent pas le code de l'algorithme, un algorithme autostabilisant retrouve de lui-même, et en temps fini, un comportement normal dès lors que les fautes transitoires ont cessé. L'autostabilisation ne fait aucune hypothèse sur la nature ou l'ampleur des fautes transitoires qui frappent le système. Ainsi, l'autostabilisation permet à un système de récupérer de l'effet de ces fautes d'une manière unifiée. De plus, l'autostabilisation comporte d'autres avantages [10]. Par exemple, un algorithme autostabilisant ne nécessite pas d'initialisation. Or, dans un système distribué, cette phase est critique, en particulier lorsque le réseau est un système à large-échelle où des milliers de nœuds peuvent être géographiquement distants. En outre, de nombreux algorithmes autostabilisants, notamment les algorithmes de calcul de tables de routage, tolèrent une certaine dynamicité du réseau au cours de l'exécution, à condition que cette dynamicité soit non silencieuse et de fréquence peu élevée. La dynamicité se définit en termes d'ajouts et/ou suppressions de processus et/ou de canaux de communication. Par dynamicité non silencieuse, nous entendons que les processus affectés par un changement topologique soient (ou finissent par être) informés de cette modification (par un protocole sous-jacent, par exemple). Par dynamicité de fréquence peu élevée, nous supposons l'existence de périodes sans changement topologique suffisamment longues pour permettre au système de stabiliser.

Cependant, l'autostabilisation a quelques inconvénients. Tout d'abord, les algorithmes autostabilisants ne masquent pas l'effet des fautes qu'ils subissent. Ainsi, les fautes transitoires provoquent une *perte de sûreté temporaire*, *i.e.*, suite à des fautes transitoires, il y a une période temporaire — appelée *phase de stabilisation* — au cours de laquelle le système n'offre aucune garantie de sûreté. De plus, dans la plupart des cas, les algorithmes autostabilisants ne permettent pas aux processeurs de détecter la fin de la phase de stabilisation. Ensuite, par définition, tout algorithme autostabilisant tolère les fautes transitoires. En revanche, la plupart des algorithmes autostabilisants ne sont pas conçus pour tolérer d'autres types de fautes, notamment lorsque les fautes sont définitives (arrêt définitif, comportement byzantin) ou intermittentes. De ce fait, la plupart des algorithmes autostabilisants deviennent totalement inopérants en présence de telles fautes. Il faut aussi noter que l'autostabilisation a un coût. Le surcoût occasionné par l'autostabilisation est observable à la fois au niveau du temps d'exécution, de l'occupation mémoire et du nombre de messages échangés. Enfin, il existe des problèmes (*e.g.*, le bit alterné [4]) pour lesquels *il n'existe pas de solution autostabilisante*.

État de l'art. Nous nous intéressons plus particulièrement ici à deux inconvénients de l'autostabilisation : la perte de sûreté temporaire et les résultats d'impossibilité.

Plusieurs spécialisations de l'autostabilisation ont été introduites pour obtenir des propriétés de sûreté plus fortes. Nous pouvons citer, par exemple, l'autostabilisation *avec contention de fautes* [6], la *super-*

stabilisation [5] ou encore la *stabilisation instantanée* (déterministe) [3]. Cette dernière est définie comme suit : dans un système réparti sujet aux pannes transitoires, un algorithme instantanément stabilisant retrouve un comportement correct *immédiatement* après la fin de celles-ci. Toutefois, un algorithme instantanément stabilisant n'offre aucune garantie pour les calculs ayant été exécutés tout ou partie durant les pannes. Bien entendu, toutes les propriétés de sûreté ne peuvent être garanties par la stabilisation instantanée. En fait, la stabilisation instantanée offre des garanties du point de vue *utilisateur* : suite à une demande de calcul de l'utilisateur, l'algorithme fournit un résultat correct vis-à-vis de la demande, quelque soit l'état dans lequel était le système lors de cette demande. Beaucoup de solutions instantanément stabilisantes sont disponibles dans la littérature. Cependant, à notre connaissance, elles sont toutes conçues pour des réseaux enracinés (où un processus est distingué) ou complètement identifiés. Nous nous intéressons ici aux réseaux *anonymes*. La stabilisation instantanée étant une spécialisation de l'autostabilisation : les résultats d'impossibilité relatifs à l'autostabilisation s'appliquent. De fait, la plupart des problèmes « classiques », comme la circulation de jeton ou encore l'élection, n'ont pas de solution autostabilisante déterministe dans les réseaux anonymes.

Plusieurs généralisations de l'autostabilisation ont été introduites pour contourner les résultats d'impossibilité, par exemple, la stabilisation faible [7], la pseudo-stabilisation [4] ou encore la stabilisation probabiliste [8]. Pour cette dernière, l'idée consiste à relâcher la notion de convergence en passant d'une convergence *déterministe* (certaine) à une convergence *probabiliste* (incertaine) : suite à des pannes transitoires, un algorithme autostabilisant probabiliste retrouve en temps *presque sûrement* fini, un comportement normal dès lors que les fautes transitoires ont cessé.

Contributions. Nous introduisons la *stabilisation instantanée probabiliste*, une généralisation de la stabilisation instantanée. L'idée principale étant de relâcher la définition de stabilisation instantanée déterministe sans altérer ses bonnes propriétés de sûreté afin d'obtenir, en particulier, des solutions instantanément stabilisantes probabilistes fonctionnant dans les réseaux anonymes.

Dans un système réparti, un algorithme instantanément stabilisant probabiliste satisfait (à nouveau) la propriété de sûreté de la tâche pour laquelle il est conçu *immédiatement* après que les pannes transitoires aient cessé ; cependant il assure la propriété de vivacité de cette tâche seulement *presque sûrement* (*i.e.*, avec probabilité 1). En cela, nous suivons une approche dite de « Las Vegas » [10].

Nous illustrons cette nouvelle propriété en proposant deux solutions, écrites dans le modèle à états (*cf.* section 2), pour le problème d'*élection avec garantie de service* fonctionnant dans un réseau anonyme, ce problème n'ayant pas de solution déterministe (stabilisante ou non). La première solution, simple, suppose que le système est synchrone. La seconde, plus complexe, fonctionne dans un système asynchrone (sous l'hypothèse la plus faible du modèle, le démon distribué inéquitable). Le problème de l'*élection avec garantie de service* consiste à assurer une réponse correcte — « oui » ou « non » — à tout processus qui pose la question « suis-je le leader du réseau ? », c'est-à-dire, (1) les réponses calculées pour chacunes des questions initiées par le même processus sont identiques ; et (2) exactement un processus reçoit toujours la réponse oui à chacune des questions qu'il initie.

Ainsi, nos algorithmes instantanément stabilisants probabilistes assurent que les réponses calculées pour chaque question initiée après que les pannes aient cessée seront toujours correctes. Cependant, le temps entre l'initialisation de la question et l'obtention de la réponse ne sera que *presque sûrement* fini.

Plan. Dans la section 2, nous décrivons brièvement le modèle dans lequel notre algorithme est écrit. Ensuite, par manque de place, nous présentons uniquement les idées principales de notre solution simple pour réseaux anonymes synchrones (section 3). Nous concluons dans la section 4.

2 Modèle

Nous considérons des réseaux bidirectionnels connexes de n processus (sans autre hypothèse topologique) où chaque processus peut communiquer directement avec un sous-ensemble d'autres processus appelés *voisins*. Les processus sont anonymes (*e.g.*, ils n'ont pas d'identité unique), cependant chaque processus peut distinguer ses voisins via un numéro de canal local.

Les processus communiquent par le biais de *variables localement partagées* : chaque processus détient un nombre fini de variables dans lesquelles il peut lire et écrire ; de plus, il peut lire les variables de ses voisins. Les variables d'un processus définissent son état. L'exécution d'un algorithme est une suite d'étapes de calcul atomiques : à chaque étape, s'il existe des processus — dits *activables* — souhaitant modifier leurs

états, alors un sous-ensemble non-vide de ces processus est activé. En une étape atomique, chaque processus activé lit ses propres variables, ainsi que celles de ses voisins, puis modifie son état. Nous supposons ici que les processus sont activés de manière synchrone, à chaque étape tous les processus activables sont activés.

Pour mesurer le temps de stabilisation, nous utilisons la notion de *ronde*, qui permet de mesurer le temps d'exécution rapporté au processus le plus lent. Ainsi, la première ronde d'une exécution termine dès lors que tous les processus continument activables depuis le début de l'exécution ont été activés au moins une fois, la seconde ronde commence et termine selon les mêmes règles, *etc*.

3 Solution synchrone

Nous détaillons maintenant le principe général de notre solution pour systèmes synchrones en adoptant une approche pas-à-pas : nous présentons d'abord un algorithme probabiliste d'élection avec garantie de service non-stabilisant ; puis, nous expliquons comment le rendre instantanément stabilisant probabiliste.

En préambule, nous avons démontré que sans aucune information globale sur le réseau (*e.g.*, une borne sur sa taille ou son diamètre), notre problème n'a pas de solution (même non-stabilisante). Nous avons obtenu ce résultat d'impossibilité en réduisant notre problème à celui du comptage des nœuds d'un anneau anonyme, ce problème étant déjà démontré insoluble [10]. Ainsi, suivant l'approche proposée dans [9], nous avons ajouté l'hypothèse suivante : une valeur B, telle que $B < n \leq 2B$, est connue de chacun des processus.

Solution probabiliste non-stabilisante. Notre algorithme exécute une infinité de *cycles*. Chaque cycle prend en entrée la valeur de la variable Booléenne ℓ de chacun des processus. Initialement, chaque processus p affecte $p.\ell$ au hasard. Dans la suite, nous appelons *candidat* tout processus p tel que $p.\ell = vrai$.

Chaque cycle a pour but d'évaluer s'il existe un unique candidat. Le résultat de cette évaluation est stocké dans la variable $Unique$ de chacun des processus. A la fin d'un cycle, les variables $Unique$ des processus ont des valeurs identiques et $Unique = vrai$ si et seulement s'il y a un unique candidat, c. Dans ce cas, c est désigné leader et les variables ℓ restent inchangées pour les cycles suivants. Dans le cas contraire, toutes variables ℓ sont réinitialisées au hasard.

Maintenant, lorsqu'un processus p initie une question (ou requête) « suis-je le leader du réseau ? », il se contente d'attendre la fin du cycle courant : si $p.Unique = vrai$, alors la réponse est $p.\ell$ et p peut renvoyer cette réponse à la couche applicative ; sinon il doit décaler la réponse à sa question au moins jusqu'à la fin du prochain cycle. Ainsi, le but est d'assurer qu'en temps presque sûrement fini il existe un unique candidat.

Nous expliquons maintenant comment les cycles sont exécutés. Initialement, $p.Unique = faux$ et $p.\ell$ est affecté au hasard par p. Ensuite, chaque cycle est divisé en 3 phases. Chaque phase doit contenir au moins D étapes de calcul, où D est le diamètre du réseau. Cela permet d'effectuer un calcul impliquant l'ensemble des processus. Ainsi, compte tenu des connaissances des processus, la longueur de chaque phase a été fixée à $2B$ ($> D$) étapes. Un cycle contient alors $6B$ étapes et est exécuté en $O(n)$ rondes (*n.b.*, dans le modèle synchrone chaque ronde correspond à une étape de calcul). Enfin, chaque processus dispose d'une horloge locale de domaine $\{0, ..., 6B-1\}$. Cette horloge est initialisée à 0 et incrémentée modulo $6B$ à chaque étape. Elle permet, en particulier, à chaque processus de savoir dans quelle phase du cycle il se trouve.

Nous détaillons maintenant chacune des phases d'un cycle. La *première phase* consiste à calculer une forêt couvrante du réseau où un processus est racine d'un arbre si et seulement s'il est candidat (s'il n'y a pas de candidats, alors aucun arbre n'existe à la fin de la phase). La première étape de calcul de cette phase consiste à initialiser, pour chaque processus, un pointeur de voisin *parent* à $\perp$ et une variable de distance d à 0. Ensuite, durant le reste de la phase, chaque processus non candidat calcule dans d sa distance au candidat le plus proche et pointe avec *parent* son voisin le plus proche de ce candidat (en cas d'égalité, le processus discrimine en utilisant les numéros de canaux locaux). Lors de la *seconde phase*, chaque candidat (s'il y en a) calcule dans sa variable entière cpt le nombre de processus appartenant à son arbre. Ce calcul s'effectue de bas en haut dans l'arbre. A la fin de la phase, si un candidat p vérifie $p.cpt > B$ [†], alors cela signifie qu'une majorité de processus appartiennent à son arbre (par définition, $B \geq \frac{n}{2}$). Il affecte alors respectivement $p.\ell$ et $p.Unique$ à vrai. Dans tous les autres cas, les processus affectent respectivement leurs variables ℓ et $Unique$ à faux. La majorité étant unique, au plus un processus est encore candidat à la fin de cette phase. La *troisième et dernière phase* consiste à propager à l'ensemble du réseau le résultat du

†. Cet évènement est réalisé en particulier si un unique processus p affecte aléatoirement $p.\ell$ à vrai au début du cycle.

cycle : à chaque étape, chaque processus réévalue sa variable $Unique$ en calculant la disjonction entre sa variable $Unique$ et celles de ses voisins. Ainsi, *à la fin du cycle*, pour chaque processus p, $p.Unique = vrai$ si et seulement s'il existe un unique candidat. Lorsqu'un unique candidat est élu, les variables ℓ deviennent constantes et les cycles suivants recalculent indéfiniment $Unique$ à vrai pour tous les processus.

La complexité (et la terminaison) de cet algorithme dépend de la loi de probabilité utilisée lorsqu'un processsus choisit une nouvelle valeur pour sa variable ℓ. Nous avons calculé que le meilleur choix possible (compte tenu des connaissances des processus) était que chaque processus p affecte $p.\ell$ à $\ll$ vrai $\gg$ avec une probabilité $Pr \in [\frac{1}{2B}, \frac{1}{B+1}]$. Dans ce cas, l'espérance du nombre de cycles à exécuter pour obtenir un unique leader est de $\frac{e^2}{2} \leq 3,70$. Ainsi, l'espérance du temps nécessaire pour calculer la réponse à une requête est en $O(n)$ rondes.

Stabilisation de la solution. L'algorithme ci-dessus fonctionne pour l'instant quand les variables sont dans un état cohérent (il n'est pas stabilisant). Cependant, le système peut se retrouver, suite à des pannes transitoires, dans une configuration quelconque où, notamment, les horloges locales des processus sont désynchronisées. Pour resynchroniser ces horloges et ainsi l'ensemble des processus, l'incrémentation des horloges est gérée par un algorithme d'unison. Nous utilisons l'algorithme autostabilisant de Boulinier *et al* [2]. Grâce à cet algorithme, les horloges locales de notre algorithme sont resynchronisées en au plus $6B$ étapes de calcul ($6B = O(n)$). Naturellement, lors du premier cycle démarré après ces $6B$ étapes de calcul, les variables $Unique$ sont réinitialisées et le résultat obtenu à la fin de ce cycle et des suivants sera correct.

Pour obtenir la stabilisation instantanée probabiliste, nous procédons alors comme suit : lorsqu'un processus p initie une question, il attend systématiquement $6B$ étapes de calcul ‡ et ne considère que les cycles démarrés après ce temps d'attente. Ainsi, il consultera uniquement les sorties (*i.e.*, $p.Unique$ et $p.\ell$) de cycles corrects ! Notez, de plus, que l'espérance du temps nécessaire pour calculer la réponse après une requête reste en $O(n)$ rondes.

4 Conclusion

Nous avons introduit la stabilisation instantanée probabiliste. Nous avons illustré cette propriété avec deux algorithmes d'élection avec garantie de service. L'un fonctionne dans un système synchrone et a une espérance de complexité en $O(n)$ rondes. L'autre (non présenté ici) fonctionne dans un système asynchrone (démon distribué inéquitable) et a une espérance de complexité en $O(n^2)$ rondes. Notez que ces complexités peuvent être réduites à $O(D)$ et $O(nD)$ si on ajoute aux processus la connaissance du diamètre du réseau.

Dans de futurs travaux, nous souhaitons proposer d'autres solutions instantanément stabilisantes probabilistes efficaces pour réseaux anonymes afin de démontrer l'intérêt et l'expressivité de cette propriété.

Références

[1] K. Altisen and S. Devismes. On probabilistic snap-stabilization. In *ICDCN*, pages 272–286, 2014.

[2] C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *PODC*, pages 150–159, 2004.

[3] A. Bui, A. K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing pif in tree networks. In *WSS*, pages 78–85. IEEE Computer Society, 1999.

[4] J. E. Burns, M. G. Gouda, and R. E. Miller. Stabilization and pseudo-stabilization. *Distributed Computing*, 7(1) :35–42, 1993.

[5] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems (abstract). In *PODC*, page 255, 1995.

[6] S. Ghosh, A. Gupta, T. Herman, and S. V. Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC*, pages 45–54, 1996.

[7] M. G. Gouda. The theory of weak stabilization. In *WSS*, pages 114–123, 2001.

[8] T. Herman. Probabilistic self-stabilization. *Inf. Proc. Letters*, 35(2) :63–67, 1990.

[9] Y. Matias and Y. Afek. Simple and efficient election algorithms for anonymous networks. In *WDAG*, pages 183–194, 1989.

[10] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

‡. Pour ce faire, il initialise un compteur à zéro lors de la requête et l'incrémente à chaque étape jusqu'à atteindre $6B$.